Software Securitv

Tackling INPUT problems

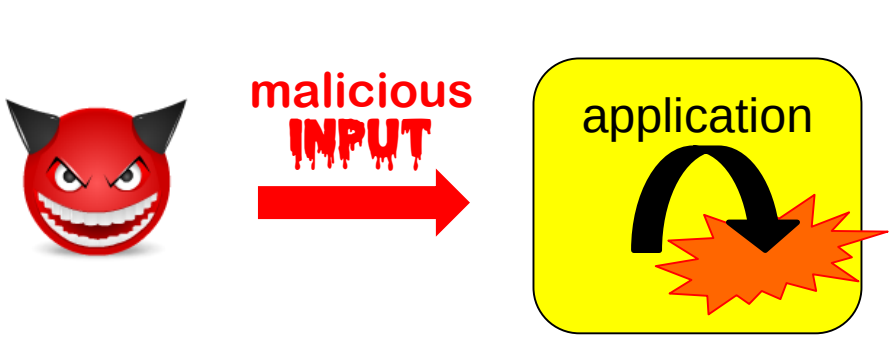
Erik Poll

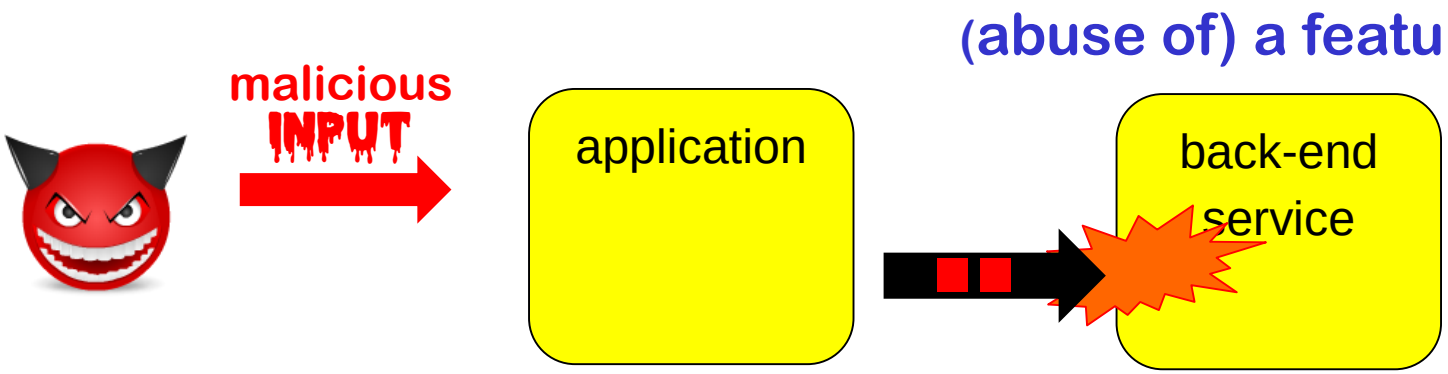
Digital Security group

Radboud University Nijmegen

TRU/e Master in Cyber Security

Recall: input attacks





Overview

Countermeasures to input attacks:

- Input validation & sanitisation
- Reducing expressive power
- Sandboxing

Input Validation,

Sanitisation,

Escaping,

Encoding,

…

Input validation aka sanitisation

- ***The*** **standard defence against malicious input**

- **'Lack of input validation' is common term for all input attacks, but this is a bit of a misnomer, in the LangSec view, as we will see later.**

- **Different ingredients:**
  1. ***How*** **to validate or sanitise?**
     a) **How to spot illegal inputs ?**
     b) **What to do with them?**
  2. ***Where*** **to validate or sanitise?**

1. Validation techniques

- **Indirect selection**
  - Let user choose from a set of legitimate inputs
  - User input never used directly by the application, and input does contaminate and taint other data
  - Most secure, but cannot be used in all situations
  - Also, attacker may be able to by-pass the user interface, eg by messing with HTTP traffic
- **White-listing**
  - List valid patterns; input *rejected unless it matches*
  - Secure, and can be used in all situations
- **Black-listing**
  - List invalid patterns; input *accepted unless it matches*
  - Least secure, given the **big** risk that some dangerous patterns are overlooked

Black-listing vs white-listing

- **Black-listing**

  Eg reject inputs that contain
  - `'` or `;` to prevent SQL injection
  - `<` or `>` to prevent HTML injection
  - `<script>` and `</script>` to prevent XSS
  - `; | < > &` to prevent OS command injection

  Warning: these blacklists are very incomplete

- **White-listing**:

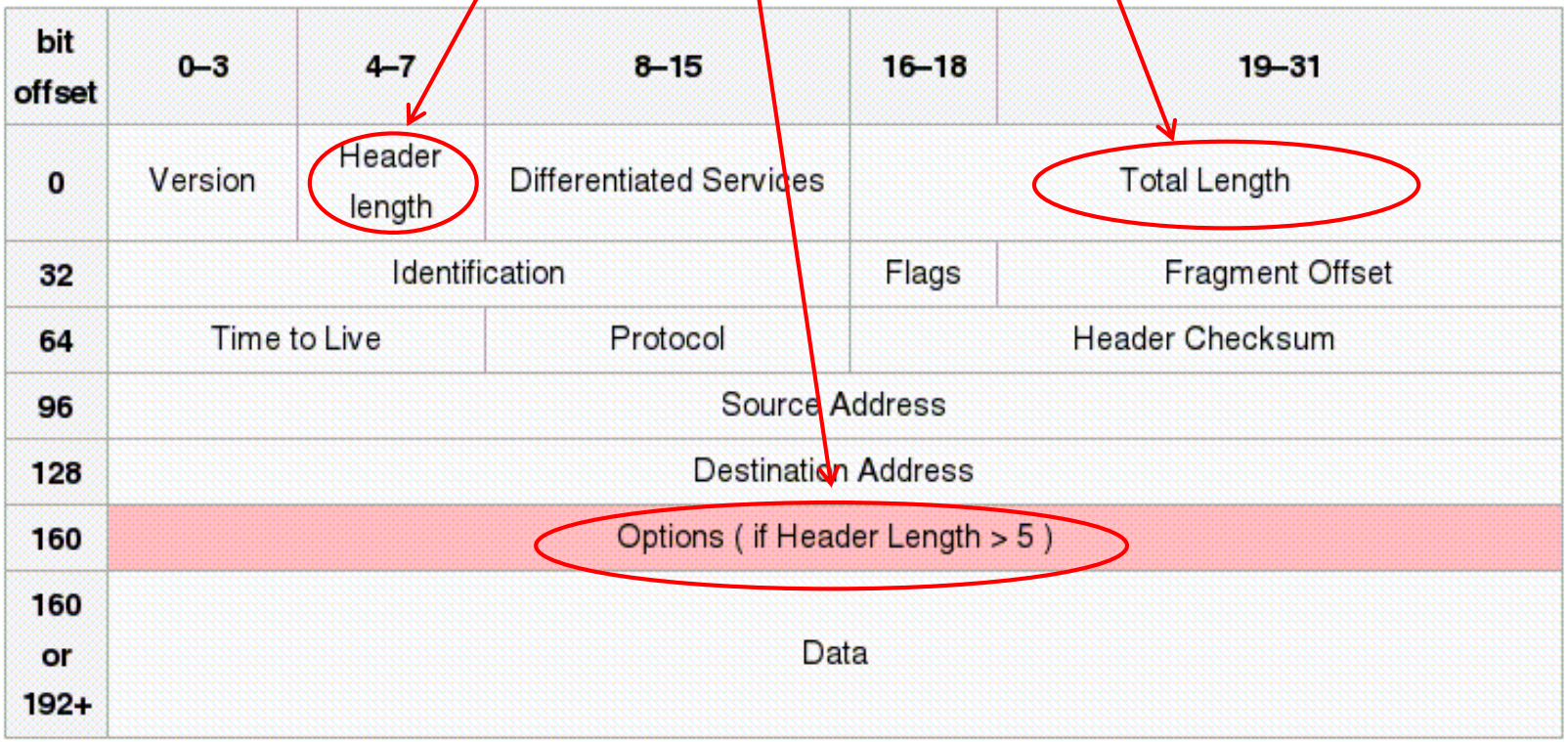
  Eg only accept inputs with `a..zA..Z0..9` to prevent SQL or HTML injection

Validation patterns

- For numbers:
  - positive, negative, max. value, possible range?
  - Or eg. Luhn mod 10 check for credit card numbers
- For strings:
  - (dis)allowed characters or words
  - More precise checks, eg using regular expressions or context-free grammars
    - Eg for RU student number (s followed by 6 digits), valid email address, URL, …
- For more complex input formats (eg Flash, JPG, PDF,…) regular expressions or grammars are not expressive enough ☹
  - Typical source of problem: length fields

Typical packet format spec

Great fun for triggering buffer overflows!



bit offset	0–3	4–7	8–15	16–18	19–31
0	Version	Header length	Differentiated Services		Total Length
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol		Header Checksum
96	Source Address				
128	Destination Address				
160	Options (if Header Length > 5)				
160 or 192+	Data				

IP packet format

Validation patterns can get COMPLEX

A regular expression to validate email adressess

```
\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
  |  "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
       |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@ (?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
  |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
       (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
          (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
          |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
     \])\z
```

This regular expression is more precise than just a whitelist of allowed characters.

See http://emailregex.com for code samples in various languages

Or read RFCs 821, 822, 1035, 1123, 2821, 2822, 3696, 4291, 5321, 5322, and 5952 and try yourself!

What to do with illegal inputs?

1. **Reject** the entire input

   with a understandable error message

2. Try to ***sanitise*** the input

   Rejecting the input is safer than trying to sanitise.

   a) ***Remove*** offending bits of the input

   b) ***Escape*** aka ***encode*** offending bits in the input

   Eg

   - replace `"` by `\"` to prevent SQL injection
   - replace `< >` by `&lt &gt` to prevent HTML/ XML injection
   - replace `script` by `xxxx` to prevent XSS
   - put quotes around some input

   NB after sanitising, changed input may need to be *re-validated*

What more to do?

Additional actions

- Log the incident
- Alert the sys-admin?

Beware of confusion

The terms

- validating
  - checking validity & rejecting - filtering out - invalid ones
- sanitising
  - somehow ‘fixing’ illegal input
- escaping
  - replacing some characters or words to sanatise input
- encoding
  - replacing all characters, eg. base64 encoding

can have slightly different but overlapping meanings, but are sometimes used interchangeably.

- Eg URL-encoding is actually a form of escaping

Canonicalisation

- **Canonicalisation**

  is the transformation of data to a unique, canonical form

  For example
  - **changing to lowercase**
  - **removing dots from the username in email address**

- **Always convert data to canonical forms**
  - ***before*** **input validation**
  - ***before*** **using it in *any* security decision**

Canonicalisation

There may be *many* ways to write the same thing, eg.

- upper or lowercase letters

  `s123456` `S123456`

- ignored characters or sub-strings

  `name+redundantstring@bla.com`

  `na.me@gmail.com` Google chooses to ignore dots in usernames

  `"Anything" name@bla.com`

  `name(some silly comment)@bla.com`

- `..` `.` `~` in path names
- file URLs `file://127.0.0.1/c|WINDOWS/clock.avi`
- using either / or \ in a URL on Windows
- URL encoding eg / encoded as `%2f`
- Unicode encoding eg / encoded as `\u002f`
- (ignored) trailing . in a domain name, eg `www.ru.nl.`
- …

Example: Complications in input validation for XSS

Many places to include javascript, and many ways to encode it, make input validation hard!

Eg

```
<script language="javascript"> alert('Hi');</script>
```

can also be written as

- `<body onload=alert('Hi')>`
- `<b onmouseover=alert('Hi')>Click here!</b>`
- `<img src="http://some.url.that/does/not/exist" onerror=alert('Hi');>`
- `<img src=j&#X41vascript:alert('Hi')>`
- `<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">`

For a longer lists of tricks, see
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Double encoding problems

Double encoding may let attackers to by-pass input validation

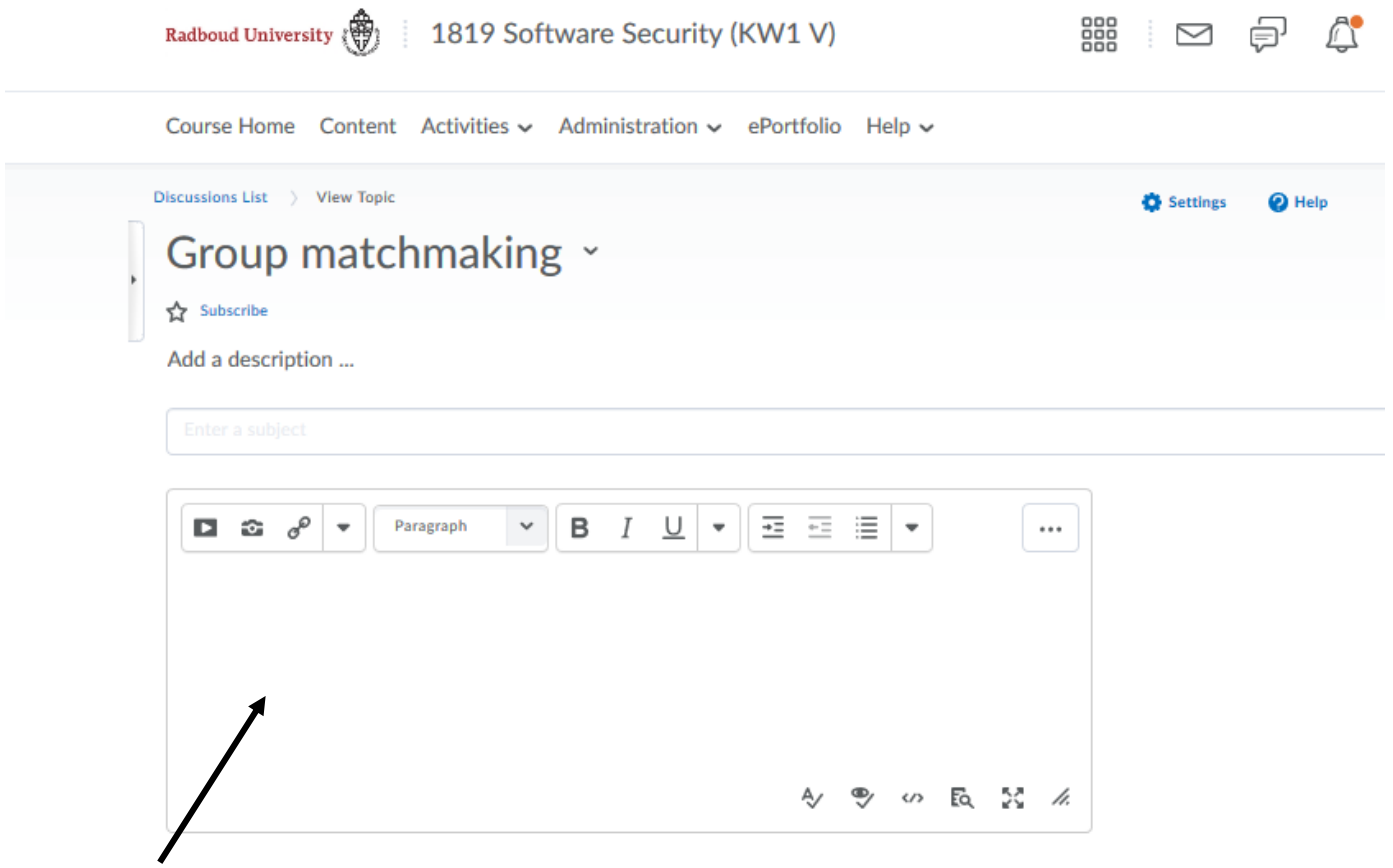
- namely if the input validation only decodes once, but an interface deeper in the application performs a second decoding

- For example, Google Chrome crashed on URL `http://%%30%30`
  - `%30` is the URL-encoding of the character `0`
  - So `%%30%30` is the URL-encoding of `%00`
  - `%00` is the URL-encoding of null character

  So `%%30%30` is a double-encoded null character

  Apparently some code deep inside Chrome does a second decoding (as a well-intended 'service' to its client code?) and then some other code chokes on the null character

Input validation nightmares



Here the user is *expected* to supply HTML…
Validating & sanitising such a rich input language is tricky!

Where to validate or sanitise?

Client- vs Server-side validation

Validation can be done client-side or server-side

- Eg, for web, in the web-browser or the web-server

Which is best? Do both of them even make sense?

Think about your attacker model!

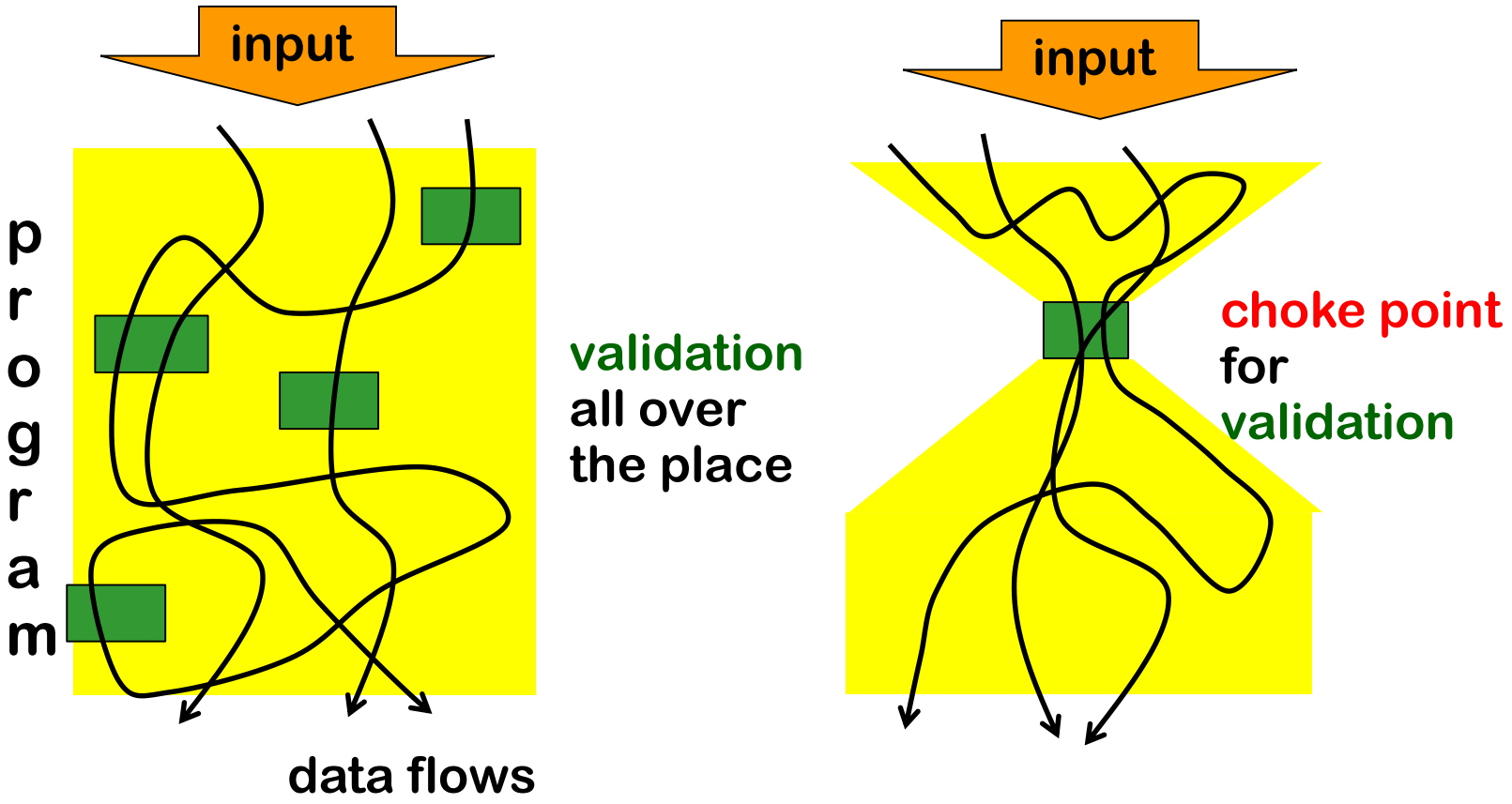
- Typically, security-critical checks must be done server-side
- Client-side checks assume the client is victim, not attacker
- Some input validation *can* or *must* be done client-side, eg
  - spotting Javascript inside a URL that a user clicks

    http://bank.com/pay.html?name=<script>.....</script>
  - in some DOM-based XSS attacks, with URLs of the form

    http://bank.com/pay.html#name=<script>.....</script>

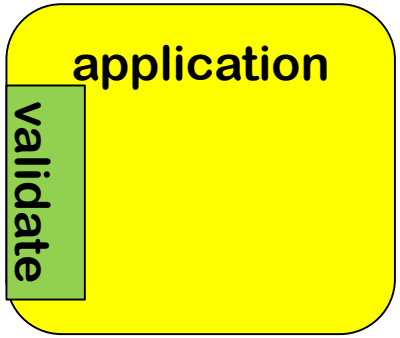
    the malicious payload stays on the client-side,
    so this can only be prevented client side

Doing validation right: at *choke points*

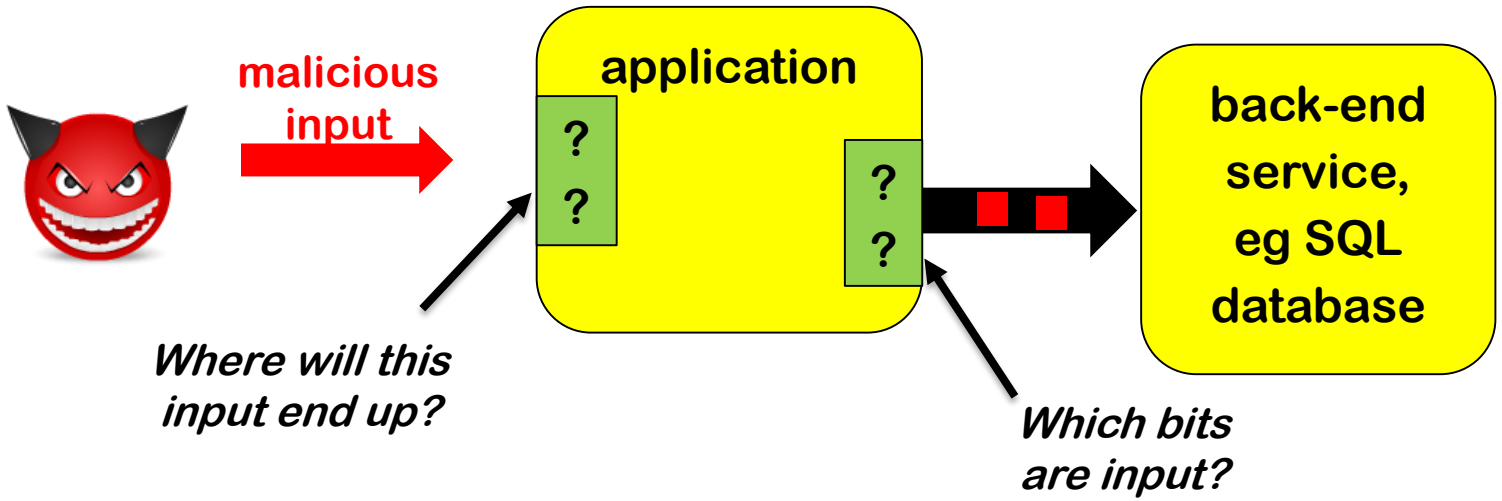


Where to validate or sanitise input?

malicious input

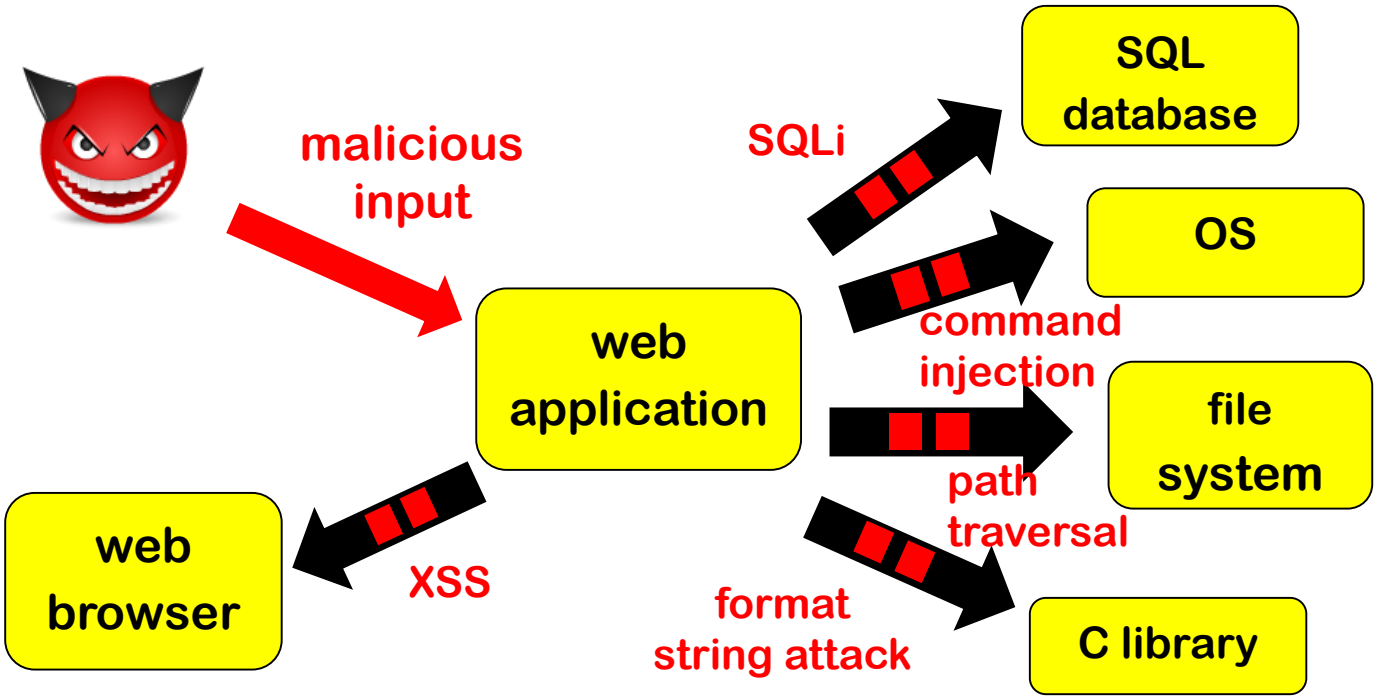


Where to validate or sanitise?



- *Rejecting illegal input* upon entry makes sense
  - eg date of birth in the future
- *Escaping dangerous input* (say because it contains `'` or `;`) less so
  - Different back-ends want different forms of escaping
    - SQL database does not like `; DROP TABLE`
      file system does not like `../../etc/passwd`
      OS does not like `& rm -fr /`

Input vs output escaping



- Output escaping make more sense than input escaping
  - because then escaping can be context-sensitive
- Downside: keeping track of which bits were input

Where & how to sanitise?

Typical combination

1. **input validation: validate input when it enters the application & reject illegal input**
2. **output sanitisation: escape output when it exits the application, eg to SQL database or OS**

- **Input sanitisation is generally a bad idea**

- **There remains a fundamental dilemma with forwarding flaws**
  - ***What*** **to validate is clearest at the *point of entry*, as there it is clear what is user input**
  - ***How*** **to escape is clearest at the *point of exit*, as there you know how the data will be used**

History of *input* sanitisation in PHP

- Function `addslashes` to escape single and double quote and null
- Magic quotes introduced in PHP2, and default in PHP3 and 4: all user parameters automatically escaped by calling `addslashes`

Why was this not a good idea?

1. different escaping needed for different SQL dialects

   eg `my_sql_real_escape_string` for MySQL

   `pg_escape_string` for PostgreSQL

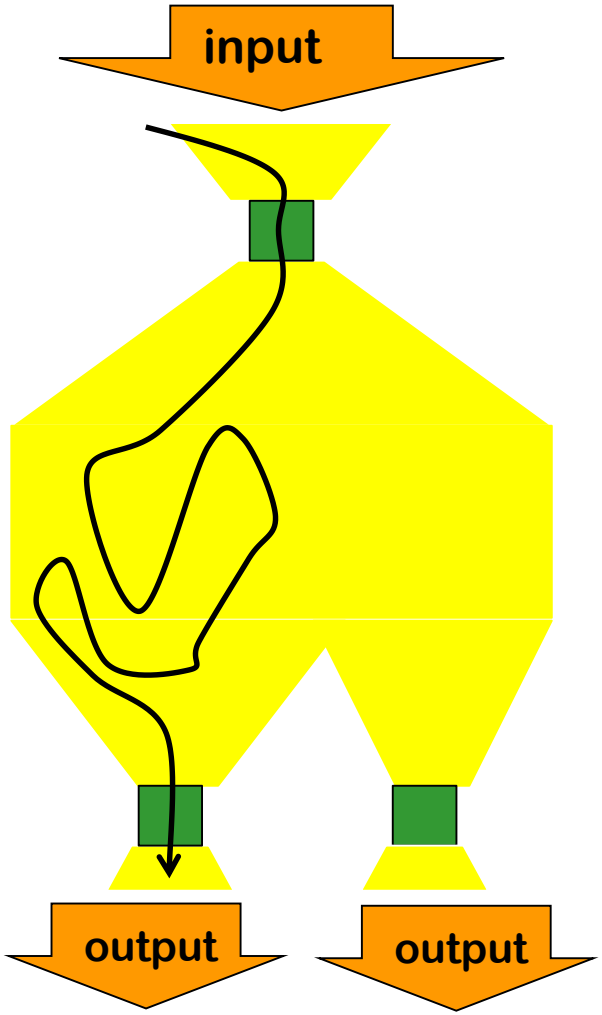
2. different escaping for different languages

   eg maybe an input needs to be escaped to prevent HTML injection, and not SQL injection?

3. giving programmer a false sense of security

- Magic quotes were removed in PHP5

chokepoints, again



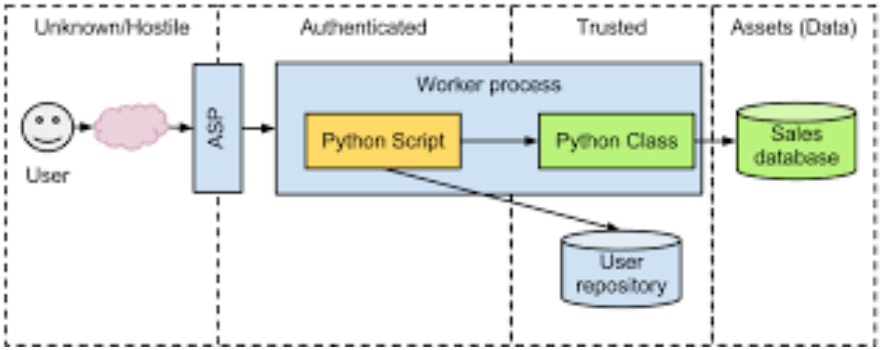
small interface
where input validation is done
close to where it enters

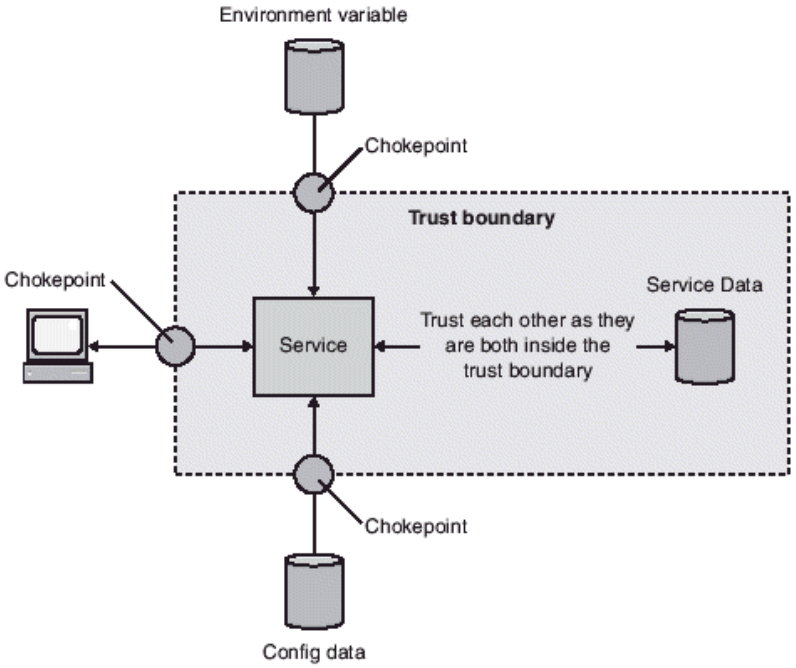
additional chokepoints
for output sanitisation

Trust-boundaries & chokepoints

Identifying trust boundary useful to decide where to validate

- **in a network, on a computer, or within an application**





But beware of data coming from trusted places, as eg. 2nd order injection attacks show

Example: 2nd order SQL injection

Suppose I want to access tanja's account

1. I register an account myself with the name `tanja' --`
2. I log in as `tanja' --` and change my password
3. If the password change is done with the SQL statement

```
UPDATE users
   SET password='abcd1234'
 WHERE username='tanja' --' and password='abc'
```

then I have reset tanja's password

- Here `abcd1234` is user input, but the dangerous input to the statement comes from the server's own database, where it was injected earlier

The moral of the story: don't trust *any* input, not even data coming from sources you think can trust

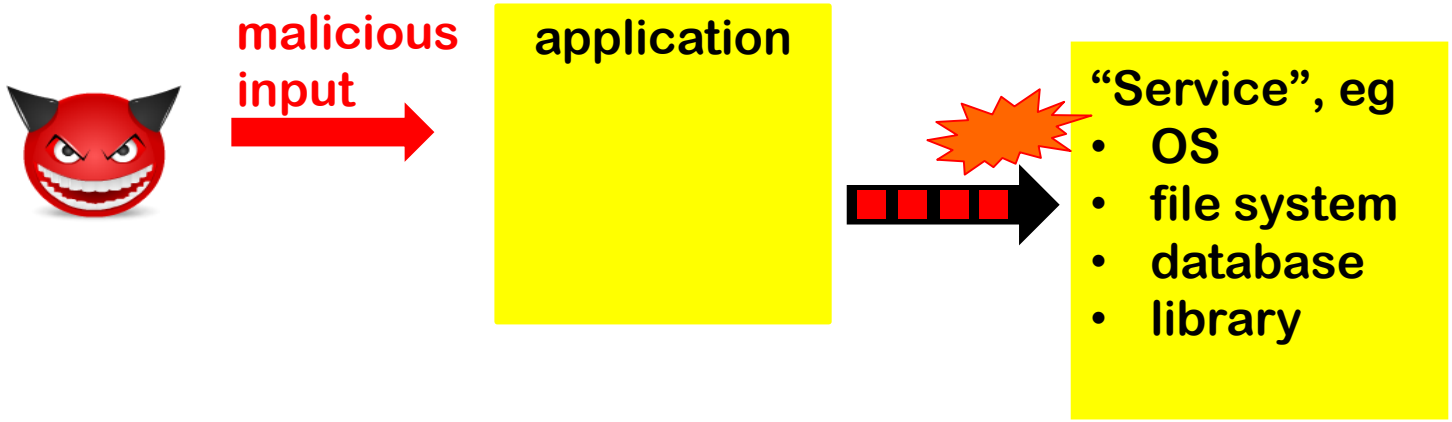
Web Application Firewall (WAF)

- A separate firewall in front of a web-application to stop malicious inputs
- Fundamental problem: *WAF has no clue what the web application is doing, and what it expects as valid inputs*
- Therefore
  - WAF can only stop very generic problems
  - To improve this, some WAFs can be trained to learn what normal inputs looks like

- *So proper input validation **still** has to done in the web application itself!*
- *Is it a useful extra line of defence? Or does it lull programmers into a false sense of security?*

Reducing expressive power

Recall forwarding flaws



The service provides a very powerful interface to the application, and hence to the attacker

- Usually, the interface takes a STRING and the service executes *any* OS command, access *any* file, execute *any* SQL command, …
- Even though the application may only requires a fraction of this power

Maybe the service should simply not offer all this power?

Prepared statements: the basic idea

Instead of a raw string as single input (aka dynamic SQL)

```
"SELECT * FROM Account WHERE Username = " + $username
                        + "AND Password = " + $password;
```

give a string with placeholders and parameters as separate inputs

```
"SELECT * FROM Account WHERE Username = ? AND Password = ?"
 $username
 $password
```

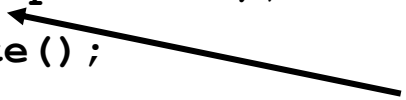
Prepared statements (aka parameterised queries)

Code vulnerable to SQL injection, using so-called dynamic SQL

```
String updateString =
  "SELECT * FROM Account WHERE Username"
   + username + "AND Password =" + password;
stmt.executeUpdate(updateString);
```

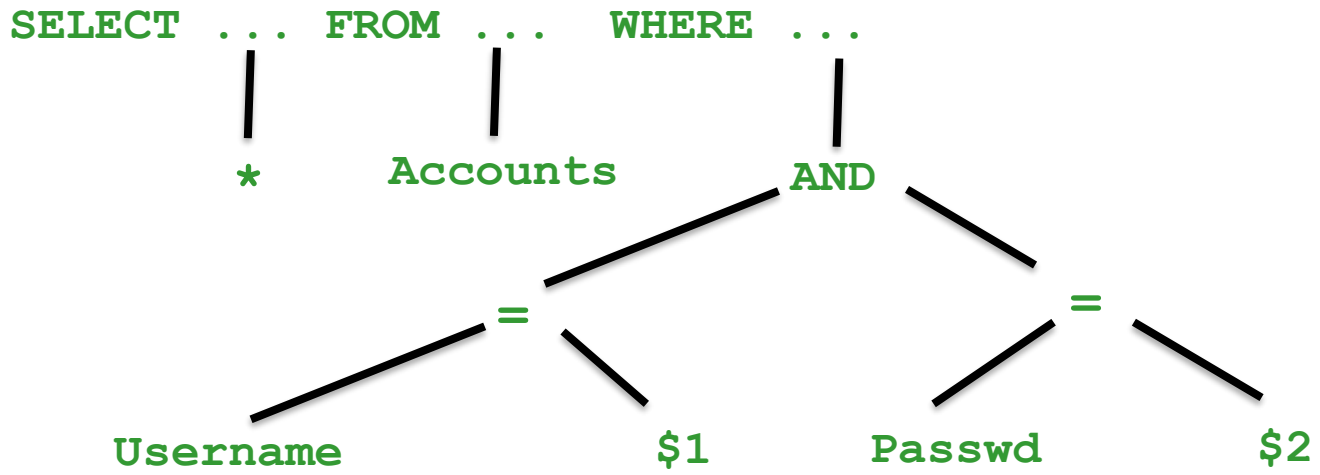
Code *not* vulnerable to SQL injection using prepared statements

```
PreparedStatement login = con.preparedStatement("SELECT
* FROM Account
    WHERE Username = ? AND Password = ?" );
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```



bind variable

The idea behind parameterised queries



- With dynamic SQL, parameters are substituted in the query string and then the result is parsed & processed
- With parameterised queries, the query is parsed *first* and and then parameters are substituted afterwards
  - The substitution then becomes less dangerous, as the impact on the meaning is reduced

Similar mechanisms

- For SQL injection: some database systems provide stored procedures.

  These *may* be safe from SQL injection, but details depend on the programming language & database system!

- For XPath injection, some APIs now offer parameterised aka pre-compiled XPath evaluation
  - eg `XPathVariableResolver` in Java

You always have to look into specific details for the combination of the programming language APIs & back-end system you use!

Example stored procedures

Stored procedure in Oracle's PL/SQL

```
CREATE PROCEDURE login
    (name VARCHAR(100), pwd VARCHAR(100)) AS
 DECLARE @sql nvarchar(4000)
 SELECT @sql =' SELECT * FROM Account WHERE
 username=' + @name + 'AND password=' + @pwd
 EXEC (@sql)
```

is safe when called from Java with

```
CallableStatement proc =
  connection.prepareCall("{call login(?, ?)}");
 proc.setString(1, username);
 proc.setString(2, password);
```

Going one step further: Wyvern

Maybe the programming language should support the various formats used (HTML, SQL, ..) as different types?

Wyvern allows such domain-specific extensions, eg

```
let authorName : String = user_input
 let webpage : HTML = ~
   <html>
     <body>
       <h1>Search results:</h1>
       <ul id="results">
         {query_results(db, ~)
           SELECT author, bookTitle FROM books
           WHERE author = {authorName}}
       </ul></body></html>
```

where HTML and SQL are different types in the language.

Tackling input language confusion

- Wyvern addresses the confusion too many input languages and formats in the programming language

- Using types or classes, similar classifications of data can be made in any (typed) programming language
  - eg using types `URL`, `EmailAdress`, `HTMLfragment`, … instead of one type `Strings` or `byte[]` for everything

- To read about Wyvern:
  Darya Kurilova, Alex Potanin, and Jonathan Aldrich, Wyvern: Impacting Software Security via Programming Language Design, PLATEAU 2014, ACM.

Sandboxing

OS sandboxing

Most basic form of sandboxing is provided by Operating System (OS) access control:

- By reducing the rights of process (or user associated with that process), we mitigate the potential damage

Counterexample:

- running your web application as root/admin

`chroot` jail

`chroot` (change root) restricts access of a process to a subset of file system, ie. changes the root of file system for that process

Eg run an application you just downloaded with

```
chroot /home/sos/erik/trial ; /tmp
```

to restrict access to just these two directories

Using the traditional OS access control permission for this, instead `chroot`, would be very tricky!

- This would require having to permissions right all over the file system

Sandboxing in browser

- JavaScript in a webpage is sand-boxed using the Same-Origin-Policy (SOP)
  - Scripts include in a webpage from A.com can only interact with content coming from A.com
  - So sub-pages (iframes) from different sources can not interact.

- Some browsers go further, and start a new OS process for every browser tab or web-domain

CSP (Content Security Policy)

CSP is a form of sandboxing implemented in browser

- **A webpage from bank.com could contain HTTP CSP header**

```
Content-Security-Policy:
default-src 'self';
img-src 'self' disney.com
child-src https://youtube.com
script-src apis.google.com
```

to *only* allow

- **images from bank.com itself or from disney.com**
- **embedded frames from youtube, included via https**
- **scripts from apis.google.com**

Warning: CSP turns out to be hard to get right!

[Weichselbaum et al., *CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy*, SIGSAC 2016]

Sandboxing for iframes

- HTML5 introduced a sandbox option to restrict what an iframe can do
- Just turning on the sandbox with no further options

  <iframe sandbox src="..."> </iframe>

  imposes many restrictions, incl.
  - no JavaScript can be executed
  - pop-up windows are blocked
  - sending of forms is blocked
  - ...
- These restrictions can be lifted one-by-one, eg

  <iframe sandbox allow-scripts allow-forms allow-pop-ups allow-same-origin src="..."> </ >
- For full list of options see
  https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox